Nitpicking C++ Concurrency

Jasmin Christian Blanchette
T. U. München, Germany
blanchette@in.tum.de

Tjark Weber
University of Cambridge, U.K.
tjark.weber@cl.cam.ac.uk

Mark Batty
University of Cambridge, U.K.
mark.batty@cl.cam.ac.uk

Scott Owens
University of Cambridge, U.K.
scott.owens@cl.cam.ac.uk

Susmit Sarkar
University of Cambridge, U.K.
susmit.sarkar@cl.cam.ac.uk

Abstract

Previous work formalized the C++ memory model in Isabelle/HOL in an effort to clarify the proposed standard's semantics. Here we employ the model finder Nitpick to check litmus test programs that exercise the memory model, including a simple locking algorithm. Nitpick is built on Kodkod (Alloy's backend) but understands Isabelle's richer logic; hence it can be applied directly to the C++ memory model. We only need to give it a few hints, and thanks to the underlying SAT solver it scales much better than the CPPMEM explicit-state model checker. This case study inspired optimizations in Nitpick from which other formalizations can now benefit.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Logic and constraint programming

General Terms Languages, Performance, Standardization

Keywords C++ memory model, concurrency, model finding, SAT solving, higher-order logic, Isabelle/HOL, Nitpick, Kodkod

1. Introduction

Most programming languages are defined by informal prose documents that contain ambiguities, omissions, and contradictions. But for many of these, researchers have constructed rigorous, mathematical semantics after the fact and formalized them in theorem provers. The benefits of formalized semantics are well known:

- They give a rigorous (and ideally readable) description of the language that can serve as a contract between designers, implementers, and users.
- They enable machine-checked proofs of theoretical results; in particular, they are an integral part of any verified compiler.
- They can be used in conjunction with lightweight formal methods, such as model checkers and model finders, to explore the consequences of the specification.

Formal methods can be quite useful for reasoning about sequential programs, but with concurrent programming they are a vital aid because of the inherent nondeterminism. Ten-line programs can have millions of possible executions. Subtle race condition bugs can remain hidden for years despite extensive testing and code reviews before they start causing failures. Even tiny concurrent programs expressed in idealized languages with clean mathematical semantics can be amazingly subtle [9, §1.4].

In the real world, performance considerations prompt hardware designers and compiler writers to further complicate the semantics of concurrent programs. For example, at the hardware level, a write operation taking place at instant t might not yet be reflected in a read at instant $t+1$ from a different thread because of cache effects. The final authority in this matter is the processor's memory consistency model.

The Java language abstracts away the various processor memory models, and compiler reordering, behind a software memory model designed to be efficiently implementable on actual hardware. However, the original Java model was found to be flawed [25], and even the revised version had some unpleasant surprises [7, 27, 29].

The next C++ standard, tentatively called C++0x, attempts to provide a clear semantics for concurrent programs, including a memory model and library functions. In previous work [3, 4], the last four authors, together with Sewell, formalized a large fragment of the prose specification in Isabelle/HOL [23] (Sects. 2 and 3). From the Isabelle formalization, they extracted the core of a tool, CPPMEM, that can check litmus test programs—small multithreaded programs that exercise various aspects of the memory model. Using this simulator, they found flaws in the original prose specification and clarified several issues, which are now addressed by the draft standard [1]. To validate the semantics, they proved the correctness of a proposed Intel x86 implementation of the concurrency primitives.

In this paper, we are interested in tool support for verifying litmus test programs. CPPMEM exhaustively enumerates the possible program executions, checking each against the (executable) formal semantics. An attractive alternative is to apply a SAT solver to the memory model constraints and litmus tests. The MemSAT tool's success on the Java memory model [29] and our early experiments [4, §6.1] suggest that SAT solvers scale better than explicit-state model checkers, allowing us to verify more complex litmus tests.

PDPP'11, July 20–22, 2011, Odense, Denmark.

Newer Isabelle versions include an efficient SAT-based model finder, Nitpick (Sect. 4). The reduction to SAT is delegated to Kodkod [28], which also serves as a backend to MemSAT and the Alloy Analyzer [13]. Nitpick and its predecessor Refute [32] featured in several case studies [5, 6, 14, 21, 31] but were, to our knowledge, never successfully applied to a specification as complex as the C++ memory model.

Although the memory model specification was not designed with SAT solving in mind, we expected that with some adjustments it should be within Nitpick's reach. The specification is written in a fairly abstract and axiomatic style, which should favor SAT solvers. Various Kodkod optimizations help cope with large problems. Moreover, although the memory model is subtle and complicated, the specification is mostly restricted to first-order logic with sets, transitive closure, and inductive datatypes, all of which are handled efficiently in Nitpick or Kodkod.

Initially, though, we had to make drastic semantics-preserving changes to the Isabelle specification so that Nitpick would scale to handle the simplest litmus tests in reasonable time (Sect. 5). These early results had been obtained at the cost of several days of work by people who understood Nitpick's internals. Based on our experience adapting the specification by hand, we proceeded to address scalability issues directly in Nitpick (Sect. 6).

With the optimizations in place, a few minor adjustments to the original memory model specification sufficed to support efficient model finding (Sect. 7). We applied the optimized version of Nitpick to several litmus tests (Sect. 8), including a simple sequential locking algorithm, thereby increasing our confidence in the specification's adequacy. Litmus tests that were previously too large for CPPMEM can now be checked within minutes.

2. Isabelle/HOL

Isabelle [23] is a generic interactive theorem prover whose built-in metalogic is a fragment of higher-order logic [2, 8, 11]. Its HOL object logic provides a more elaborate version of higher-order logic, complete with the familiar connectives and quantifiers.

The term language consists of simply-typed λ-terms augmented with constants (of scalar or function types) and ML-style polymorphism. Function application expects no parentheses around the argument list and no commas in between, as in $f\,x\,y$. Syntactic sugar provides an infix syntax for common operators, such as $x = y$ and $x + y$. Variables may range over functions and predicates. Types are usually implicit but can be specified using a constraint $:: \tau$.

HOL's standard semantics interprets the Boolean type *bool* and the function space $\sigma \to \tau$. The function arrow associates to the right, reflecting the left-associativity of application. Predicates are functions of type $\sigma_1 \to \cdots \to \sigma_n \to bool$. HOL identifies sets with monadic predicates and provides syntactic sugar for set notations.

Inductive datatypes can be defined by specifying the constructors and the types of their arguments. The type *nat* of natural numbers is defined by the command

datatype *nat* = 0 | *Suc nat*

The type is generated freely from the constructors $0 :: nat$ and $Suc :: nat \to nat$. The polymorphic type α *list* of lists over α is defined as

datatype α *list* = *Nil* | *Cons* α (α *list*)

Since lists are so common, Isabelle also supports the more convenient notation $[x_1, \ldots, x_n]$ for $Cons\ x_1\ (\ldots(Cons\ x_n\ Nil)\ldots)$.

Constants can be introduced by free-form axioms or, more safely, by a simple definition in terms of existing constants. In addition, Isabelle provides high-level definitional principles for inductive and coinductive predicates as well as recursive functions.

3. C++ Memory Model

The C++ Final Draft International Standard [1] defines the concurrency memory model axiomatically, by imposing constraints on the order of memory accesses in program executions. The semantics of a C++ program is then a set of allowed executions.

Here we briefly present the memory model and its Isabelle/HOL formalization [3, 4], focusing on the aspects that are necessary to understand the rest of this paper.

3.1 Introductory Example

To facilitate efficient implementations on modern parallel architectures, the C++ memory model (like other relaxed memory models) has no global linear notion of time. Program executions are not guaranteed to be *sequentially consistent* (*SC*)—that is, equivalent to a simple interleaving of threads [18].

The following program fragment is a simple example that can exhibit non-SC behavior:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
    printf("y: %d\n", y.load(ord_relaxed));
|||
    y.store(1, ord_relaxed);
    printf("x: %d\n", x.load(ord_relaxed));
}}}
```

(To keep examples simple, we use the notation `{{{ … ||| … }}}` for parallel composition and `ord_xxx` for `memory_order_xxx`.)

Two threads write to separate memory locations `x` and `y`; then each thread reads from the other location. Can both threads read the original value of the location they read from? According to the draft standard, they can. The program has eight outputs that are permitted by the memory model:

x: 0 y: 0	y: 0 x: 0	x: 0 y: 1	y: 0 x: 1	x: 1 y: 0	y: 1 x: 0	x: 1 y: 1	y: 1 x: 1

Among these, the first two outputs exhibit the counterintuitive non-SC behavior.

3.2 Memory Actions and Orderings

From the memory model's point of view, C++ program executions consist of *memory actions*. The main actions are

$\mathsf{L}\ \mathtt{x}$	locking of `x`
$\mathsf{U}\ \mathtt{x}$	unlocking of `x`
$\mathsf{R}_{ord}\ \mathtt{x} = v$	atomic read of value v from `x`
$\mathsf{W}_{ord}\ \mathtt{x} = v$	atomic write of value v to `x`
$\mathsf{RMW}_{ord}\ \mathtt{x} = v_1/v_2$	atomic read–modify–write at `x`
$\mathsf{R}_{\mathsf{na}}\ \mathtt{x} = v$	nonatomic read of value v from `x`
$\mathsf{W}_{\mathsf{na}}\ \mathtt{x} = v$	nonatomic write of value v to `x`
F_{ord}	fence

where *ord*, an action's *memory order*, can be any of the following:

sc	`ord_seq_cst`	sequentially consistent
rel	`ord_release`	release
acq	`ord_acquire`	acquire
a/r	`ord_acq_rel`	acquire and release
con	`ord_consume`	consume
rlx	`ord_relaxed`	relaxed

Memory orders control synchronization and ordering of atomic actions. The `ord_seq_cst` ordering provides the strongest guarantees (SC), while `ord_relaxed` provides the weakest guarantees. The release/acquire discipline, where writes use `ord_release`

and reads use `ord_acquire`, occupies an intermediate position on the continuum. The slightly weaker variant release/consume, with `ord_consume` for the reads, can be implemented more efficiently on hardware with weak memory ordering.

If we wanted to prohibit the non-SC executions in the program above, we could simply pass `ord_seq_cst` as argument to the `load` and `store` functions instead of `ord_relaxed`.

3.3 Isabelle/HOL Formalization

In place of a global timeline, the standard postulates several relations over different subsets of a program's memory actions. These relations establish some weak notion of time. They are not necessarily total or transitive (and can therefore be hard to understand intuitively) but must satisfy various constraints.

In the Isabelle/HOL formalization of the memory model [3, 4], a candidate execution $\mathcal{X}$ is a pair $(\mathcal{X}_{\mathsf{opsem}}, \mathcal{X}_{\mathsf{witness}})$. The component $\mathcal{X}_{\mathsf{opsem}}$ specifies the program's memory actions (*acts*), its threads (*thrs*), a simple typing of memory locations (*lk*), and four relations over actions (*sb*, *asw*, *dd*, and *cd*) that constrain their evaluation order. The other component, $\mathcal{X}_{\mathsf{witness}}$, consists of three relations: Read actions *read from* some write action (*rf*), *sequentially consistent* actions are totally ordered (*sc*), and a *modification order* (*mo*) gives a per-location linear-coherence order for atomic writes.

$\mathcal{X}_{\mathsf{opsem}}$ is given by the program's operational semantics and can be determined statically from the program source. $\mathcal{X}_{\mathsf{witness}}$ is existentially quantified. The C++ memory model imposes constraints on the components of $\mathcal{X}_{\mathsf{opsem}}$ and $\mathcal{X}_{\mathsf{witness}}$ as well as on various relations derived from them. A candidate execution is *consistent* if it satisfies these constraints. The Isabelle/HOL top-level definition of consistency is given below:

```
definition consistent_execution acts thrs lk sb asw dd cd
                                rf mo sc ≡
  well_formed_threads acts thrs lk sb asw dd cd ∧
  well_formed_reads_from_mapping acts lk rf ∧
  consistent_locks acts thrs lk sb asw dd cd sc ∧
  let
    rs   = release_sequence acts lk mo
    hrs  = hypothetical_release_sequence acts lk mo
    sw   = synchronizes_with acts sb asw rf sc rs hrs
    cad  = carries_a_dependency_to acts sb dd rf
    dob  = dependency_ordered_before acts rf rs cad
    ithb = inter_thread_happens_before acts thrs lk sb asw
                                       dd cd sw dob
    hb   = happens_before acts thrs lk sb asw dd cd ithb
    vse  = visible_side_effect acts thrs lk sb asw dd cd hb
  in
    consistent_inter_thread_happens_before acts ithb ∧
    consistent_sc_order acts thrs lk sb asw dd cd mo sc hb ∧
    consistent_modification_order acts thrs lk sb asw dd cd
                                  sc mo hb ∧
    consistent_reads_from_mapping acts thrs lk sb asw dd
                                  cd rf sc mo hb vse
```

The derived relations (*rs*, *hrs*, etc.) and the various consistency conditions follow the C++ final draft standard; we omit their definitions. The complete memory model comprises approximately 1200 lines of Isabelle text.

3.4 CPPMEM

For any given $\mathcal{X}_{\mathsf{opsem}}$, there may be one, several, or perhaps no choices for $\mathcal{X}_{\mathsf{witness}}$ that give rise to a consistent execution. Since the memory model is complex, and the various consistency conditions and their interactions can be difficult to understand intuitively, tool support for exploring the model and computing the possible behaviors of C++ programs is much needed.

The CPPMEM tool [4] was designed to assist with these tasks. It consists of the following three parts: (1) a preprocessor that computes the $\mathcal{X}_{\mathsf{opsem}}$ component of a candidate execution from a C++ program's source code; (2) a search procedure that enumerates the possible values for $\mathcal{X}_{\mathsf{witness}}$; (3) a checking procedure that calculates the derived relations and evaluates the consistency conditions for each pair $(\mathcal{X}_{\mathsf{opsem}}, \mathcal{X}_{\mathsf{witness}})$.

For the second part, we refrained from implementing a sophisticated memory-model-aware search procedure in favor of keeping the code simple. CPPMEM enumerates all possible combinations for the *rf*, *mo*, and *sc* relations that respect a few basic constraints:

- *sc* only contains SC actions and is a total order over them.
- *mo* only contains pairs (a,b) such that a and b write to the same memory location; for each location, *mo* is a total order over the set of writes at this location.
- *rf* only contains pairs (a,b) such that a writes a given value to a location and b reads the same value from that location; for each read b, it contains exactly one such pair.

Because the search space grows asymptotically with $n!$ in the worst case, where n is the number of actions in the program execution, CPPMEM is mostly limited to small litmus tests, which typically involve up to eight actions. This does cover many interesting tests, but not larger parallel algorithms.

Writing a more sophisticated search procedure would require a detailed understanding of the memory model (which we hope to gain through proper tool support in the first place) and could introduce errors that are difficult to detect—unless, of course, the procedure was automatically generated from the formal specification. This is where Nitpick comes into play.

4. Nitpick

Nitpick [6] is a model finder for Isabelle/HOL based on Kodkod [28], a constraint solver for first-order relational logic (FORL) that in turn relies on the SAT solver MiniSat [10]. Given a conjecture, Nitpick searches for a standard set-theoretic model that satisfies the given formula as well as any relevant axioms and definitions. Isabelle users can invoke the tool manually at any point in an interactive proof to find models or countermodels. Unlike Isabelle itself, which adheres to the LCF "small kernel" discipline [12], Nitpick does not certify its results and must be trusted.

Nitpick's design was inspired by its predecessor Refute [32], which performs a direct reduction to SAT. The translation from HOL is parameterized by the cardinalities of the atomic types occurring in it [5]. Nitpick systematically enumerates the cardinalities, so that if the formula has a finite model, the tool eventually finds it, unless it runs out of resources.

Given finite cardinalities, the translation to FORL is straightforward, but common HOL idioms require a translation scheme tailored for SAT solving. In particular, infinite datatypes are soundly approximated by subterm-closed finite substructures [16] axiomatized in a three-valued logic.

Example. The Isabelle/HOL formula $\mathit{rev}\ xs = xs \wedge |\mathit{set}\ xs| = 2$ specifies that the list xs is a palindrome involving exactly two distinct elements. When asked to provide a model, Nitpick almost instantly finds $xs = [a_1, a_2, a_1]$. The detailed output reveals that it approximated the type *α list* with the finite substructure $\{[], [a_1], [a_2], [a_1, a_2], [a_2, a_1], [a_1, a_2, a_1]\}$.

5. First Experiments

In early experiments briefly treated elsewhere [4, §6.1], we tried out Nitpick on a previous version of the Isabelle formalization of the C++ memory model (the "post-Rapperswil model" [4]) to check

whether given pairs $(\mathcal{X}_{\mathsf{opsem}}, \mathcal{X}_{\mathsf{witness}})$ are consistent executions of litmus test programs—a task that takes CPPMEM only a few milliseconds. Nitpick printed "Nitpicking…" but then became entangled seemingly forever in its problem-generation phase, not ever reaching Kodkod or MiniSat. What was happening?

5.1 Constant Unfolding

The first issue, a fairly technical one, was that the constant unfolding code was out of control. Nitpick has a simplistic approach to constants occurring in a satisfaction problem:

- For simple definitions $c \equiv t$, Nitpick unfolds (i.e., inlines) the constant c's definition, substituting the right-hand side t for c wherever it appears in the problem.
- Constants introduced using **primrec** or **fun**, which define recursive functions in terms of user-supplied equational specifications in the style of functional programming, are translated to FORL variables, and their equational specifications are conjoined with the problem as additional constraints to satisfy.

This approach works reasonably well for specifications featuring a healthy mixture of simple definitions and recursive functions, but it falls apart when there are too many nested simple definitions, as is the case in the C++ memory model. Simple definitions are generally of the form $c \equiv (\lambda x_1 \ldots x_n.\, u)$, and when c is applied to arguments, these are substituted for $x_1, \ldots, x_n$. The formula quickly explodes if the x_i's occur many times in the body u.

Since simple definitions are a degenerate form of recursion, we added hints to the specification telling Nitpick to treat some of the simply defined constants as if they had been recursive functions. In addition, we used the 'let' construct to store the value of an argument that is needed several times. HOL 'let' bindings are translated to an analogous FORL construct, enabling subexpression sharing all the way down to the SAT problem [28, §4.3].

5.2 Higher-Order Arguments

When we invoked Nitpick again, it produced worrisome warning messages such as the following one:

Arity 23 too large for universe of cardinality 4.

This specific message indicates that one of the free variables in the generated Kodkod problem is a 23-ary relation, whose cardinality might exceed $2^{31} - 1$ and cause arithmetic overflow in Kodkod. Nitpick detects this and warns the user.

Alloy users are taught to avoid relations of arities higher than 3 because they are very expensive in SAT: A relation over A^n requires $|A|^n$ propositional variables. HOL n-ary functions are normally translated to $(n+1)$-ary relations. Absurdly high arities arise when higher-order arguments are translated. A function from σ to τ cannot be directly passed as an argument in FORL; the workaround is to pass $|\sigma|$ arguments of type τ that encode a function table.

The memory model specification defines many constants with higher-order signatures, mainly because HOL identifies sets with their characteristic predicates. Here is one example among many:

$$\begin{aligned}&\mathit{synchronizes\_with} ::\\ &\quad (\mathit{act} \to \mathit{bool}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to \mathit{act} \to \mathit{act} \to \mathit{bool}\end{aligned}$$

An important optimization in Nitpick, function specialization, eliminates the most superficial higher-order arguments [6, §5.1]. A typical example is f in the definition of *map*:

$$\begin{aligned}&\textsf{primrec}\ \mathit{map}\ \textsf{where}\\ &\mathit{map}\ f\ \mathit{Nil} = \mathit{Nil}\\ &\mathit{map}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs})\end{aligned}$$

Nitpick will specialize the *map* function for each admissible call site, thereby avoiding passing the argument for f altogether. At the call site, any argument whose free variables are all globally free (or are bound but less expensive to pass than the argument itself) is eligible for this optimization.

Most of the higher-order arguments in the memory model specification were eliminated this way. The remaining higher-order arguments resulted from a bad interaction between specialization and 'let'. Specifically, each 'let' introduces a higher-order bound variable, which prevents the argument from being specialized. For example, the crucial *consistent_execution* predicate's definition comprises the following 'let' bindings:

$$\begin{aligned}&\mathit{rs} = \mathit{release\_sequence}\ \mathit{acts}\ \mathit{lk}\ \mathit{mo}\\ &\mathit{hrs} = \mathit{hypothetical\_release\_sequence}\ \mathit{acts}\ \mathit{lk}\ \mathit{mo}\\ &\mathit{sw} = \mathit{synchronizes\_with}\ \mathit{acts}\ \mathit{sb}\ \mathit{asw}\ \mathit{rf}\ \mathit{sc}\ \mathit{rs}\ \mathit{hrs}\end{aligned}$$

The first two variables bound are of type $\mathit{act} \to \mathit{act} \to \mathit{bool}$. When these variables are passed to *synchronizes_with*, each of them is encoded as $|\mathit{act}|^2$ arguments of type *bool*. The easiest solution is to unfold higher-order 'let' bindings. This enables specialization to perform the essential work of keeping relation arities low, at the cost of some duplication in the generated FORL formula.

There remained one troublesome higher-order function in connection with so-called visible sequences of side-effects:

$$\begin{aligned}&\mathit{visible\_sequences\_of\_side\_effects\_set} ::\\ &\quad (\mathit{act} \to \mathit{bool}) \to (\mathit{thr\_id} \to \mathit{bool}) \to\\ &\quad (\mathit{loc} \to \mathit{loc\_kind}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad (\mathit{loc} \to \mathit{loc\_kind}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad (\mathit{loc} \to \mathit{loc\_kind}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad (\mathit{loc} \to \mathit{loc\_kind}) \to (\mathit{act} \times \mathit{act} \to \mathit{bool}) \to\\ &\quad \mathit{act} \times (\mathit{act} \to \mathit{bool}) \to \mathit{bool}\end{aligned}$$

The issue here is the return type $\mathit{act} \times (\mathit{act} \to \mathit{bool}) \to \mathit{bool}$: a set of pairs whose second components are sets of actions—effectively, a set of sets. We eventually found a way to completely eliminate visible sequences of side-effects without affecting the specification's semantics, as explained in Sect. 7.

5.3 Datatype Spinning

Having resolved the other problems, there remained one issue with Nitpick's handling of inductive datatypes. The memory model specification defines a few datatypes, notably a type *act* of actions:

$$\begin{aligned}&\textsf{datatype}\ \mathit{act} =\\ &\quad \mathit{Lock} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{loc}\\ &\mid \mathit{Unlock} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{loc}\\ &\mid \mathit{Atomic\_load} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{ord\_order}\ \mathit{loc}\ \mathit{val}\\ &\mid \mathit{Atomic\_store} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{ord\_order}\ \mathit{loc}\ \mathit{val}\\ &\mid \mathit{Atomic\_rmw} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{ord\_order}\ \mathit{loc}\ \mathit{val}\ \mathit{val}\\ &\mid \mathit{Load} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{loc}\ \mathit{val}\\ &\mid \mathit{Store} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{loc}\ \mathit{val}\\ &\mid \mathit{Fence} \quad \mathit{act\_id}\ \mathit{thr\_id}\ \mathit{ord\_order}\end{aligned}$$

The specification manipulates it through discriminators and selectors. An example of each follows:

$$\begin{aligned}&\textsf{definition}\ \mathit{is\_store}\ a \equiv\\ &\quad \text{case}\ a\ \text{of}\ \mathit{Store}\ \_\ \_\ \_\ \_ \Rightarrow \mathit{True} \mid \_ \Rightarrow \mathit{False}\end{aligned}$$

$$\begin{aligned}&\textsf{fun}\ \mathit{thread\_id\_of}\ \textsf{where}\\ &\mathit{thread\_id\_of}\ (\mathit{Lock}\ \_\ \mathit{tid}\ \_) = \mathit{tid}\\ &\mathit{thread\_id\_of}\ (\mathit{Unlock}\ \_\ \mathit{tid}\ \_) = \mathit{tid}\\ &\quad \vdots\\ &\mathit{thread\_id\_of}\ (\mathit{Fence}\ \_\ \mathit{tid}\ \_) = \mathit{tid}\end{aligned}$$

Because of the type's high (in fact, infinite) cardinality, Nitpick cannot assign a distinct FORL atom to each possible *act* value. Instead,

it considers only a subset of the possible values and uses a special undefined value, denoted by $\star$, to stand for the other values. Constructors and other functions sometimes return $\star$. Undefined values trickle down from terms all the way to the logical connectives and quantifiers. This leads to a Kleene three-valued logic [5, §4.2].

Which subset of *act* values should Nitpick choose? Perhaps surprisingly, it makes no specific commitment beyond fixing a cardinality for the set that approximates *act* (by default, 1, 2, and so on up to 10). It is the SAT solver's task to exhaust the possible subsets. The SAT solver finds out automatically which subset is needed. For example, if we state the conjecture

$$\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys} \longrightarrow \mathit{xs} = \mathit{ys}$$

on lists, Nitpick finds the counterexample $\mathit{xs} = [a_1]$ and $\mathit{ys} = [a_2]$ with values taken from the subset $\{[], [a_1], [a_2]\}$. (The empty list is needed to build the other two lists.) Contrast this with

$$\mathit{length}\ \mathit{xs} < 2$$

where Nitpick instead relies on the subset $\{[], [a_1], [a_1, a_1]\}$.

This SAT-based approach to datatypes follows an established Alloy idiom [16] that typically scales up to cardinality 8 or so. However, in larger specifications such as the C++ memory model, the combinatorial explosion goes off much earlier.

And yet, for our specification, this combinatorial spinning is unnecessary: The needed actions all appear as ground terms in the litmus tests. The memory model specification inspects the actions, extracting subterms and recombining them in sets, but it never constructs new actions. Unfortunately, neither Nitpick nor Kodkod notices this, and MiniSat appears to be stumped.

Once again, we found ourselves modifying the specification to enforce the desired behavior. We first replaced the *act* datatype with an enumeration type that lists all the actions needed by a given litmus test and only those. For example:

datatype $\mathit{act} = a \mid b \mid c \mid d \mid e \mid f$

Then we defined the discriminators and selectors appropriately for the litmus test of interest. For example:

fun *is_store* **where**
$\mathit{is\_store}\ a = \mathit{True}$
$\mathit{is\_store}\ \_ = \mathit{False}$

fun *thread_id_of* **where**
$\mathit{thread\_id\_of}\ f = 1$
$\mathit{thread\_id\_of}\ \_ = 0$

Using this idiom, we could reuse the bulk of the memory model specification text unchanged across litmus tests. The main drawback of this approach is that each litmus test requires its own definitions for the *act* datatype and the discriminator and selector functions, on which the memory model specification rests.

The Nitpick-based empirical results presented in our previous paper [4, §6.1] employed this approach. The largest litmus test we considered (IRIW-SC) required 5 minutes using CPPMEM, but only 130 seconds using Nitpick. Other examples took Nitpick about 5 seconds. These were one-off experiments, since further changes to the original memory model were not mirrored in the Nitpick-enabled specification. The experiments were nonetheless interesting in their own right and persuaded us to optimize Nitpick further.

6. New Nitpick Optimizations

The experiments described in the previous section shed some light on areas in Nitpick that could benefit from more optimizations. Although we had come up with acceptable workarounds, we decided to improve the tool in the hope that future similar applications would require less manual work.

6.1 Heuristic Constant Unfolding

Nitpick's constant unfolding behavior, described in Sect. 5.1, left much to be desired. The notion that by default every simple definition should be unfolded was clearly misguided. If a constant is used many times and its definition is large, it should most likely be kept in the translation to facilitate reuse. Kodkod often detects shared subterms and factors them out [28, §4.3], but it does not help if Nitpick's translation phase is overwhelmed by the large terms.

We introduced a simple heuristic based on the size of the right-hand side of a definition: Small definitions are unfolded, whereas larger ones are kept as equations. It is difficult to design a better heuristic because of hard-to-predict interactions with function specialization and other optimizations occurring at later stages. We also made the unfolding more compact by heuristically introducing 'let's to bind the arguments of a constant when it is unfolded.

Nitpick provides a *nitpick_simp* hint that can be attached to a simple definition or a theorem of the right form to prevent unfolding. We now added a *nitpick_unfold* hint that can be used to forcefully unfold a larger definition, to give the user complete control.

6.2 Necessary Datatype Values

In Sect. 5.3 we explained how Nitpick's subterm-closed subset approach to inductive datatypes leads to a combinational explosion in the SAT solver. We also introduced an Isabelle idiom to prevent this "datatype spinning" when we know which values are necessary. However, the idiom requires fundamental changes to the specification, which is highly undesirable if proving and code generation (for CPPMEM) must also be catered for.

We came up with a better plan: Add an option to let users specify necessary datatype values and encode that information in the Kodkod problem. Users provide a set of datatype values as ground constructor terms, and Nitpick assigns one FORL atom to each term and subterm from that set. The atom assignment is coded as an additional constraint in the FORL problem and passed to the SAT solver, which exploits it to prune the search space.

For checking litmus tests, a convenient idiom is to first declare all the actions required as Isabelle abbreviations. For example:

abbreviation $a \equiv \mathit{Atomic\_store}\ 0\ 0\ \mathit{Ord\_relaxed}\ 0\ 0$
abbreviation $b \equiv \mathit{Atomic\_rmw}\ 1\ 0\ \mathit{Ord\_relaxed}\ 0\ 0\ 1$
$\vdots$
abbreviation $f \equiv \mathit{Atomic\_rmw}\ 5\ 1\ \mathit{Ord\_release}\ 0\ 2\ 3$

If we pass the option "$\mathit{need} = a\ b\ c\ d\ e\ f$", Nitpick then generates additional constraints for these six terms and also their subterms:

$\mathsf{a}_1 = \mathit{Atomic\_store}\ 0\ 0\ \mathsf{a}_7\ 0\ 0$
$\mathsf{a}_2 = \mathit{Atomic\_rmw}\ 1\ 0\ \mathsf{a}_7\ 0\ 0\ 1$
$\vdots$
$\mathsf{a}_6 = \mathit{Atomic\_rmw}\ 5\ 1\ \mathsf{a}_8\ 0\ 2\ 3$
$\mathsf{a}_7 = \mathit{Ord\_relaxed}$
$\mathsf{a}_8 = \mathit{Ord\_release}$

Natural numbers are left alone. The optimization makes no sense for them, since the desired cardinality k fully determines the subterm-closed subset to use, namely $\{0, \mathit{Suc}\ 0, \ldots, \mathit{Suc}^{k-1}\ 0\}$.

The resulting performance behavior is essentially the same as with the idiom presented in Sect. 5.3, but without having to restructure the specification.

We expect this optimization to be generally useful for specifications of programming language semantics. We had previously undertaken unpublished experiments with a Java compiler verified in Isabelle [19] and found that Nitpick suffered heavily from datatype spinning: Even when the program was hard-coded in the problem, the SAT solver would needlessly spin through the subterm-closed substructures of all possible Java programs.

6.3 Two-Valued Translation

There is a second efficiency issue related to Nitpick's handling of inductive datatypes. As mentioned in Sect. 5.3, attempting to construct a value that Nitpick cannot represent yields the unknown value $\star$. The ensuing partiality is handled by a Kleene three-valued logic, which is expressed in terms of Kodkod's two-valued logic as follows: At the outermost level, the FORL truth value true encodes *True* (genuine model), whereas false stands for both *False* (no model) and $\star$ (potentially spurious model). The same convention is obeyed in other positive contexts within the formula (i.e., under an even number of negations). Dually, false encodes *False* in negative contexts and true encodes *True* or $\star$.

This approach is incomplete but sound: When Kodkod finds a (finite) FORL model, it always corresponds to a (possibly infinite) HOL model. Unfortunately, the three-valued logic puts a heavy burden on the translation, which must handle $\star$ gracefully and keep track of polarities (positive and negative contexts). An operation as innocuous as equality in HOL, which we could otherwise map to FORL equality, must be translated so that it returns $\star$ if either operand is $\star$.

Formulas occurring in unpolarized, higher-order contexts (e.g., in a conditional expression or as argument to a function) are less common but all the more problematic. The conditional

$$\text{if } \varphi \text{ then } t_1 \text{ else } t_2$$

is translated to

$$\mathsf{if}\ \widehat{\varphi}^{+}\ \mathsf{then}\ \widehat{t}_1\ \mathsf{else\ if}\ \neg\,\widehat{\varphi}^{-}\ \mathsf{then}\ \widehat{t}_2\ \mathsf{else}\ \star$$

where $\widehat{\varphi}^{+}$ is the translation of φ in a positive context and $\widehat{\varphi}^{-}$ is its negative counterpart. The translation can quickly explode in size if φ itself contains formulas in higher-order places.

Simply stated, the root of the problem is overflow: When a constructor returns $\star$, it overflows in much the same way that IEEE n-bit floating-point arithmetic operations can yield NaN ("not a number") for large operands. Nitpick's translation to FORL detects overflows and handles them soundly. If the tool only knew that overflows are impossible, it could disable this expensive machinery and use a more direct two-valued translation [5, §4.1].

With the subterm-closed substructure approximation of inductive datatypes, overflows are a fact of life. For example, the append operator @ on lists, which is specified by the equations

$$\begin{aligned} \mathit{Nil} \mathbin{@} \mathit{ys} &= \mathit{ys} \\ (\mathit{Cons}\ x\ \mathit{xs}) \mathbin{@} \mathit{ys} &= \mathit{Cons}\ x\ (\mathit{xs} \mathbin{@} \mathit{ys}) \end{aligned}$$

will overflow if the longest representable nonempty list is appended to itself. In contrast, selectors such as *hd* (head) and *tl* (tail), which return the first and second argument of a *Cons*, cannot overflow.

The C++ memory model is a lengthy specification consisting of about 1200 lines of Isabelle/HOL definitions. On the face of it, it would be most surprising that no overflows are possible anywhere in it. But this is essentially the case. Apart from the test program and the set of possible executions, which appear in the conjecture and which we treated specially in Sect. 6.2, no overflows can occur. The whole specification is a predicate that merely inspects $\mathcal{X}_{\mathsf{opsem}}$ and $\mathcal{X}_{\mathsf{witness}}$ without constructing new values. Coincidentally, this design ensures the fast execution of CPPMEM, which must check thousands of potential witnesses.

To exploit this precious property of the specification, we added an option, *total_consts*, to control whether the two-valued translation without $\star$ should be used. The two-valued translation must be used with care: A single overflow in a definition can prevent the discovery of models.

Naturally, it would be desirable to implement an analysis in Nitpick to determine precisely which constants can overflow and use this information in a hybrid translation. This is future work.

7. Fine-Tuned C++ Memory Model

In parallel with our work on Nitpick, we refined the specification of the memory model in three notable ways:

- We ported the model to the custom specification language LEM; we now generate Isabelle text from that [24]. Ideally, we want to apply Nitpick directly on the generated specification.
- Reflecting recent discussions in the C++ concurrency subcommittee, in the new model SC reads cannot read from non-SC writes that happened before SC writes. This minor technical change rules out certain counterintuitive executions.
- The new model is formulated in terms of *visible side-effects* (a relation over actions) but does without the more complicated notion of *visible sequences of side-effects* (a relation between actions and sets of actions) found in the draft standard. This allows all consistency conditions to be predicates over sets and binary relations, which reduces Nitpick's search space considerably. We recently completed a formal equivalence proof for the two formulations.

With the optimizations described in Sect. 6 in place, Nitpick handles the new specification reasonably efficiently without any modifications. It nonetheless pays off to fine-tune the specification in three respects.

First, the types *act_id*, *thr_id*, *loc*, and *val*—corresponding to action IDs, thread IDs, memory locations, and values—are defined as aliases for *nat*, the type of natural numbers. This is unfortunate because it prevents us from specifying different cardinalities for the different notions. A litmus test involving eight actions, five threads, two memory locations, and two values gives rise to a much smaller SAT problem if we tell Nitpick to use the cardinalities $|\mathit{act\_id}| = 8$, $|\mathit{thr\_id}| = 5$, and $|\mathit{loc}| = |\mathit{val}| = 2$ than if all four types are set to have cardinality 8. To solve this, we replace the aliases

$$\begin{aligned} &\textbf{type\_synonym}\ \mathit{act\_id} = \mathit{nat} \\ &\textbf{type\_synonym}\ \mathit{thr\_id} = \mathit{nat} \\ &\textbf{type\_synonym}\ \mathit{loc} \quad = \mathit{nat} \end{aligned}$$

in the LEM specification with copies of the natural numbers:

$$\begin{aligned} &\textbf{datatype}\ \mathit{act\_id} = \mathit{A0} \mid \mathit{ASuc}\ \mathit{act\_id} \\ &\textbf{datatype}\ \mathit{thr\_id} = \mathit{T0} \mid \mathit{TSuc}\ \mathit{thr\_id} \\ &\textbf{datatype}\ \mathit{loc} \quad = \mathit{L0} \mid \mathit{LSuc}\ \mathit{loc} \end{aligned}$$

For notational convenience, we define the following abbreviations: $a_k \equiv \mathit{ASuc}^k\ \mathit{A0}$, $t_k \equiv \mathit{TSuc}^k\ \mathit{T0}$, $\mathtt{x} \equiv \mathit{L0}$, and $\mathtt{y} \equiv \mathit{LSuc}\ \mathit{L0}$.

Second, while the unfolding heuristic presented in Sect. 6.1 allowed us to remove many *nitpick_simp* hints, we found that the heuristic was too aggressive with respect to two constants, which are better unfolded (using *nitpick_unfold*). We noticed them because they were the only relations of arity greater than 3 in the generated Kodkod problem.[1] Both were called with a higher-order argument that was not eligible for specialization. We specified the *nitpick_unfold* hints in a separate theory file that imports the LEM-generated Isabelle specification and customizes it.

Third, some of the basic definitions in the specification are gratuitously inefficient for SAT solving. The specification had its own definition of relational composition and transitive closure, but it is preferable to replace them with equivalent concepts from Isabelle's libraries, which are mapped to appropriate FORL constructs. This is achieved by providing lemmas that redefine the memory model's constants in terms of the desired Isabelle concepts:

[1] One could expect that the *act* constructors, which take between 3 and 6 arguments each, would be mapped to relations of arities 4 to 7, but Nitpick encodes constructors in terms of selectors to avoid high-arity relations [5, §5.2]. A term such as *Cons x xs* is translated to "the nonempty list *ys* such that $\mathit{hd}\ \mathit{ys} = x$ and $\mathit{tl}\ \mathit{ys} = \mathit{xs}$," where *hd* and *tl* are coded as binary relations.

lemma [*nitpick_unfold*]: *compose R S* $= R \circ S$
lemma [*nitpick_unfold*]: *tc A R* $= (\mathit{restrict\_relation}\ R\ A)^+$

These lemmas are part of the separate theory file mentioned above. Similarly, we supply a more compact definition of the predicate *strict_total_order_over A R*. The original definition cleanly separates the constraints expressing the relation's domain, irreflexivity, transitivity, and totality:

$$\forall (a,b) \in R.\ a \in A \wedge b \in A$$
$$\forall x \in A.\ (x,x) \notin R$$
$$\forall x \in A.\ \forall y \in A.\ \forall z \in A.\ (x,y) \in R \wedge (y,z) \in R \longrightarrow (x,z) \in R$$
$$\forall x \in A.\ \forall y \in A.\ (x,y) \in R \vee (y,x) \in R \vee x = y$$

The optimized formulation

$$\big(\forall x\ y.\ \text{if } (x,y) \in R \text{ then}$$
$$\quad \{x,y\} \subseteq A \wedge x \neq y \wedge (y,x) \notin R$$
$$\text{else}$$
$$\quad \{x,y\} \subseteq A \wedge x \neq y \longrightarrow (y,x) \in R\big) \wedge$$
$$R^+ = R$$

reduces the number of occurrences of A and R, both of which are higher-order arguments that are instantiated with arbitrarily large terms by the specialization optimization.

8. Litmus Tests

We evaluated Nitpick on several litmus tests designed to illustrate the semantics of the C++ memory model. The experiments were conducted on a 64-bit Mac Pro system with a Quad-Core Intel Xeon processor at 2.66 GHz clock speed, exploiting a single core. They are discussed in more detail below.

Most of these litmus tests had been checked by CPPMEM and the unoptimized version of Nitpick before [4], so it should not come as a surprise that our experiments revealed no further flaws in the C++ final draft standard. We did, however, discover many mistakes in the latest version of the formalization, such as missing parentheses and typos (e.g., $\forall$ instead of $\exists$). These mistakes had been accidentally introduced when the model was ported to LEM and had gone unnoticed even though it is used as a basis for formal proofs. Our experience illustrates once again the need to validate complex specifications, to ensure that the formal artifact correctly captures the intended semantics of the informal one (in our case, the draft standard).

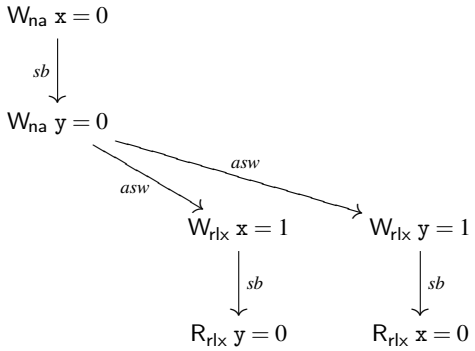
8.1 Store Buffering

Our first test is simply the introductory example from Sect. 3:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
    printf("y: %d\n", y.load(ord_relaxed));
|||
    y.store(1, ord_relaxed);
    printf("x: %d\n", x.load(ord_relaxed));
}}}
```

This program has six actions: two nonatomic initialization writes ($\mathsf{W_{na}}$), then one relaxed write ($\mathsf{W_{rlx}}$) and one relaxed read ($\mathsf{R_{rlx}}$) in each thread. The diagram below shows the relations *sb* and *asw*, which are part of $\mathcal{X}_{\mathsf{opsem}}$ and hence fixed by the program's operational semantics:



Each vertex represents an action, and an r-edge from a to b indicates that $(a,b) \in r$. The actions are arranged in three columns corresponding to the threads they belong to. For transitive relations, we omit transitive edges.

To check a litmus test with Nitpick, we must provide $\mathcal{X}_{\mathsf{opsem}}$. We can use CPPMEM's preprocessor to compute $\mathcal{X}_{\mathsf{opsem}}$ from a C++ program's source code, but for simple programs such as this one we can also translate the code manually. We first declare abbreviations for the test's actions:

abbreviation $a \equiv$ *Store* $a_0\ t_0$ x 0
abbreviation $b \equiv$ *Store* $a_1\ t_0$ y 0
abbreviation $c \equiv$ *Atomic_store* $a_2\ t_1$ *Ord_relaxed* x 1
abbreviation $d \equiv$ *Atomic_load* $a_3\ t_1$ *Ord_relaxed* y 0
abbreviation $e \equiv$ *Atomic_store* $a_4\ t_2$ *Ord_relaxed* y 1
abbreviation $f \equiv$ *Atomic_load* $a_5\ t_2$ *Ord_relaxed* x 0

Notice that the read actions, *d* and *f*, specify the value they expect to find as last argument to the constructor. That value is 0 for both threads because we are interested only in non-SC executions, in which the write actions *c* and *e* are ignored by the reads.

Next, we introduce the components of $\mathcal{X}_{\mathsf{opsem}}$ as constants:

definition [*nitpick_simp*]: *acts* $\equiv \{a,b,c,d,e,f\}$
definition [*nitpick_simp*]: *thrs* $\equiv \{t_0,t_1,t_2\}$
definition [*nitpick_simp*]: *lk* $\equiv (\lambda\_.\ \mathit{Atomic})$
definition [*nitpick_simp*]: *sb* $\equiv \{(a,b),(c,d),(e,f)\}$
definition [*nitpick_simp*]: *asw* $\equiv \{(b,c),(b,e)\}$
definition [*nitpick_simp*]: *dd* $\equiv \{\}$
definition [*nitpick_simp*]: *cd* $\equiv \{\}$

Specialization implicitly propagates these values to where they are needed in the specification. To avoid clutter and facilitate subexpression sharing, we disable unfolding by specifying *nitpick_simp*.

Finally, we look for a model satisfying the constraint

consistent_execution acts thrs lk sb asw dd cd rf mo sc

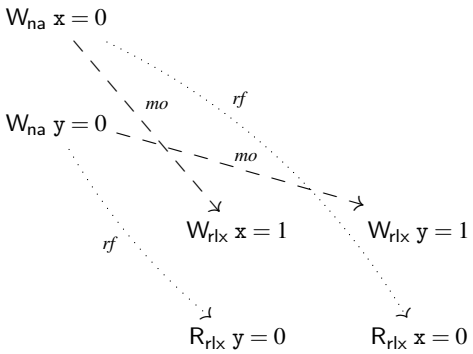
where *rf*, *mo*, and *sc* are free variables corresponding to $\mathcal{X}_{\mathsf{witness}}$. The Nitpick call is shown below:

nitpick [

satisfy,	look for a model
need = *a b c d e f*,	the necessary actions (Sect. 6.2)
card act = 6,	six actions (a,b,c,d,e,f)
card act_id = 6,	six action IDs $(a_0,a_1,a_2,a_3,a_4,a_5)$
card thr_id = 3,	three thread IDs (t_0,t_1,t_2)
card loc = 2,	two locations (x, y)
card val = 2,	two values $(0,1)$
card = 10,	maximum cardinality for other types
total_consts,	use two-valued translation (Sect. 6.3)
finitize act,	pretend *act* is finite
dont_box	disable boxing [6, §5.1]

]

With these options, Nitpick needs 4.7 seconds to find relations that witness a non-SC execution:



The modification-order relation (*mo*) reveals that the assignments $x = 1$ and $y = 1$ take place after the initializations to 0, but the read-from relation (*rf*) indicates that the two reads get their values from the initializations and not from the assignments.

If we replace all four relaxed memory accesses with SC atomics, such non-SC behavior is no longer possible. Nitpick verifies this in 4.0 seconds by reporting the absence of suitable witnesses. These results are consistent with our understanding of the draft standard.

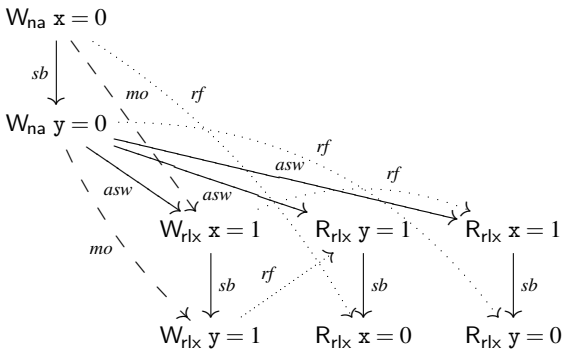
8.2 Message Passing

We consider a variant of message passing where one thread writes to a data location `x` and then a flag `y`, while two reading threads read both the flag and the data. There are two initialization writes and two actions in each thread, for a total of eight actions:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
    y.store(1, ord_relaxed);
|||
    printf("y1: %d\n", y.load(ord_relaxed));
    printf("x1: %d\n", x.load(ord_relaxed));
|||
    printf("x2: %d\n", x.load(ord_relaxed));
    printf("y2: %d\n", y.load(ord_relaxed));
}}}
```

Because all non-initialization actions are relaxed atomics, it is possible for the two readers to observe the writes in opposite order. Nitpick finds the following execution in 5.7 seconds:



On the other hand, if the flag is accessed via a release/acquire pair, or via SC atomics, the first reader is guaranteed to observe the data written by the writer thread. Nitpick finds such an execution (without the second reader) in 4.0 seconds, and verifies the absence of a non-SC execution also in 4.0 seconds.

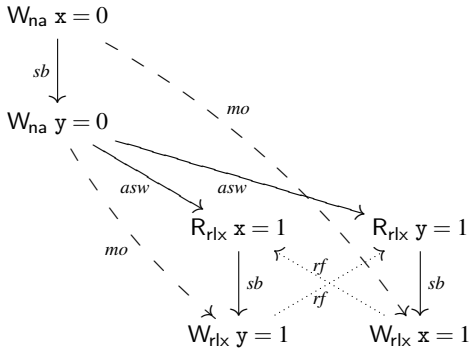
8.3 Load Buffering

This test is dual to the Store Buffering test. Two threads read from separate locations; then each thread writes to the other location:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    printf("x: %d\n", x.load(ord_relaxed));
    y.store(1, ord_relaxed);
|||
    printf("y: %d\n", y.load(ord_relaxed));
    x.store(1, ord_relaxed);
}}}
```

With relaxed atomics, each thread can observe the other thread's later write. Nitpick finds a witness execution in 4.2 seconds:



Nitpick verifies the absence of a non-SC execution with release/consume, release/acquire, and SC atomics in 4.1 seconds.

8.4 Sequential Lock

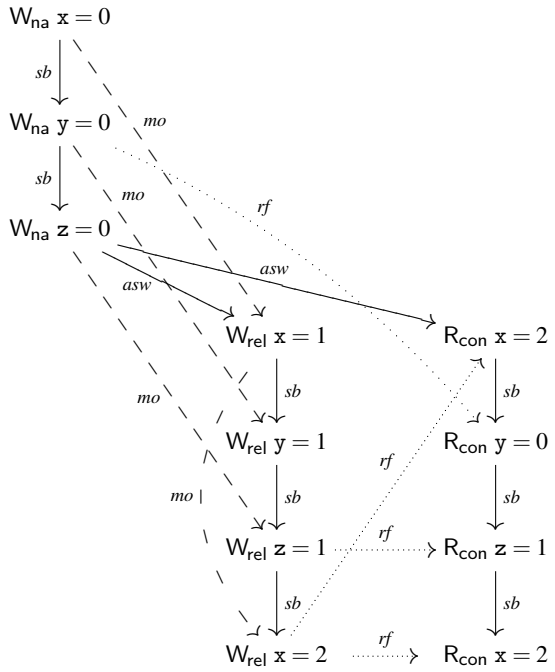
This test is more ambitious. The program models a simple sequential locking algorithm inspired by the Linux kernel's "seqlock" mechanism [17]:

```
atomic_int x = 0;
atomic_int y = 0;
atomic_int z = 0;

{{{
    for (int i = 0; i < N; i++) {
        x.store(2 * i + 1, ord_release);
        y.store(i + 1, ord_release);
        z.store(i + 1, ord_release);
        x.store(2 * i + 2, ord_release);
    }
|||
    printf("x: %d\n", x.load(ord_consume));
    printf("y: %d\n", y.load(ord_consume));
    printf("z: %d\n", z.load(ord_consume));
    printf("x: %d\n", x.load(ord_consume));
}}}
```

The program spawns a writer and a reader thread. The writer maintains a counter `x`; the writer's loop increments it before and after modifying the data locations `y` and `z`. Intuitively, the data is "locked" whenever the counter `x` is odd. The reader eventually accesses the data, but not without checking `x` before and after. A non-SC behavior occurs if the two reads of `x` yield the same even value (i.e., the lock was free during the two data reads) but $y \neq z$ (i.e., the data was observed while in an inconsistent state).

Already for $\mathtt{N} = 1$, Nitpick finds the following non-SC execution, where the reader observes $\mathtt{y} = 0$ and $\mathtt{z} = 1$, in 15.8 seconds:



With release/acquire instead of release/consume, the algorithm should be free of non-SC behavior. Nitpick takes 15.8 seconds to exhaustively check the $\mathtt{N} = 1$ case, 86 seconds for $\mathtt{N} = 2$, and 378 seconds for $\mathtt{N} = 3$. If we add a second reader thread, it takes 86 seconds for $\mathtt{N} = 1$ and 379 seconds for $\mathtt{N} = 2$.

Because of the loop, our analysis is incomplete: We cannot prove the absence of non-SC behavior for all bounds `N` (or for an arbitrary number of readers), only its presence. Nonetheless, the small-scope hypothesis, which postulates that "most bugs have small counterexamples" [13, §5.1.3], strongly suggests that the sequential locking algorithm implemented in terms of release/acquire atomics is correct for any number of iterations and reader threads.

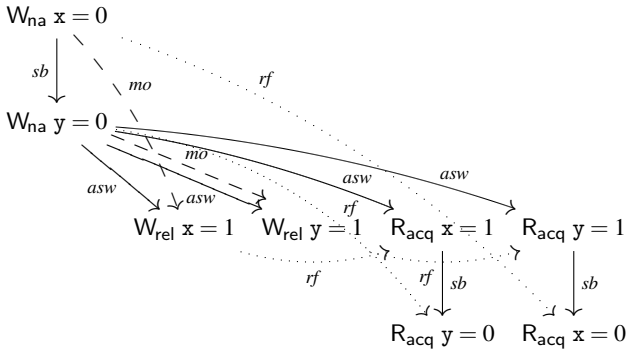
8.5 Independent Reads of Independent Writes

Two writer threads independently write to `x` and `y`, and two readers read from both locations:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_release);
|||
    y.store(1, ord_release);
|||
    printf("x1: %d\n", x.load(ord_acquire));
    printf("y1: %d\n", y.load(ord_acquire));
|||
    printf("y2: %d\n", y.load(ord_acquire));
    printf("x2: %d\n", x.load(ord_acquire));
}}}
```

With release/acquire, release/consume, and relaxed actions, different reader threads can observe these writes in opposite order. Nitpick finds an execution in 5.8 seconds:



With SC actions, this behavior is not allowed, and Nitpick verifies the absence of a non-SC execution in 5.2 seconds.

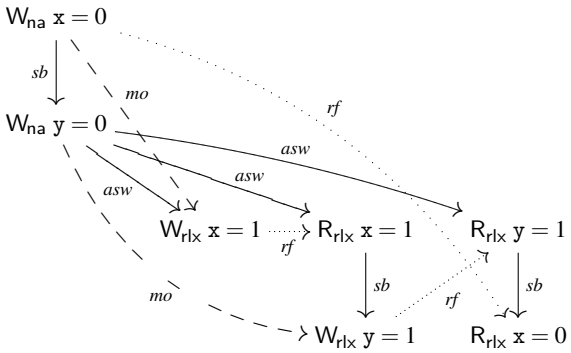
8.6 Write-to-Read Causality

This test spawns three auxiliary threads in addition to the implicit initialization thread:

```
atomic_int x = 0;
atomic_int y = 0;

{{{
    x.store(1, ord_relaxed);
|||
    printf("x1: %d\n", x.load(ord_relaxed));
    y.store(1, ord_relaxed);
|||
    printf("y: %d\n", y.load(ord_relaxed));
    printf("x2: %d\n", x.load(ord_relaxed));
}}}
```

The first auxiliary thread writes to `x`; the second thread reads from `x` and writes to `y`; the third thread reads from `y` and then from `x`. With relaxed atomics, the third thread does not necessarily observe the first thread's write to `x` even if it observes the second thread's write to `y` and the second thread observes the first thread's write to `x`. Nitpick finds this execution in 4.2 seconds:



The memory model guarantees write-to-read causality for release/acquire and SC actions. Nitpick verifies the absence of a non-SC execution in 4.4 seconds.

8.7 Generalized Write-to-Read Causality

Nitpick's run-time depends on the size of the search space, which is exponential in the number of actions. To demonstrate how Nitpick scales up to larger litmus tests, we generalize the Write-to-Read Causality test from 2 to n locations. The generalized test consists of $2n$ writes (including n initializations) and $n+1$ reads, thus $3n+1$ actions in total. Since the three witness variables are binary relations over actions, the state space is of size $2^{3(3n+1)^2}$.

With relaxed atomics, there is an execution where the last thread does not observe the first thread's write. With SC atomics, no such execution exists. The Nitpick run-times are tabulated below. For comparison, we also include the CPPMEM run-times (on roughly comparable hardware).

Locations (n)	Actions ($3n+1$)	States ($2^{3(3n+1)^2}$)	CPPMEM relaxed	CPPMEM SC	Nitpick relaxed	Nitpick SC
2	7	2^{147}	0.0 s	0.5 s	4 s	4 s
3	10	2^{300}	0.0 s	90.5 s	11 s	11 s
4	13	2^{507}	0.1 s	$>10^4$ s	41 s	40 s
5	16	2^{768}	0.2 s	$>10^4$ s	132 s	127 s
6	19	2^{1083}	0.7 s	$>10^4$ s	384 s	376 s
7	22	2^{1452}	2.5 s	$>10^4$ s	982 s	977 s

Each additional location slows down Nitpick's search by a factor of about 3. Although the search space grows with 2^{n^2}, the search time grows slightly slower than k^n, which is asymptotically better than CPPMEM's $n!$ worst-case complexity.

CPPMEM outperforms Nitpick on the relaxed version of the test because its basic constraints reduce the search space to just 2^n candidate orders for *rf* and *mo* (Sect. 3.4). On the other hand, CPPMEM scales much worse than Nitpick when the actions are SC, because it naively enumerates all $2^n \cdot (2n+1)!$ combinations for *rf*, *mo*, and *sc* that meet the basic constraints.

8.8 Further Remarks

Thanks to the optimizations presented in Sect. 6, Nitpick is about 25 times faster on medium-sized litmus tests than it was before. Verifying the absence of a consistent non-SC execution for the Independent Reads of Independent Writes test now takes about 5.2 seconds, compared with 130 seconds previously and 5 minutes using CPPMEM [4, §6.1]. Larger SC tests that cannot realistically be checked with CPPMEM are now analyzable within minutes.

On small litmus tests, Nitpick remains significantly slower than CPPMEM, which takes less than a second on some of the tests. The bottleneck is the translation of the memory model into FORL and SAT. The SAT search is extremely fast for small tests and scales much better than CPPMEM's simplistic enumeration scheme. On the largest problems we considered, Nitpick takes a few seconds, which is negligible; then about 95% of the time is spent in Kodkod, while the rest is spent in MiniSat.

For some litmus tests, CPPMEM's basic constraints reduce the search space considerably. We could probably speed up Nitpick by incorporating these constraints into the model—for example, by formalizing *rf* as a map from reads to writes, rather than as a binary relation over actions. However, this would require extensive modifications to the formalization, which we would rather avoid.

9. Related Work

The discovery of fatal flaws in the original Java memory model [25] stimulated much research in software memory models. We attempt to cover the most relevant work, focusing on tool support.

MemSAT [29] is an automatic tool based on Kodkod specifically designed for debugging axiomatic memory models. It has been used on several memory models from the literature, including the Java memory model. A noteworthy feature of MemSAT is that it produces a minimal unsatisfiable core if the model or the litmus test is overconstrained. MemSAT also includes a component that generates relational constraints from Java programs, akin to CPPMEM's preprocessor. Nitpick's main advantage over MemSAT for our case study is that it understands higher-order logic.

NemosFinder [33] is another axiomatic memory model checker. Memory models are coded as Prolog predicates and either checked using constraint logic programming or SAT solving. The tool includes a specification of the Intel Itanium memory model.

Visual-MCM [22] is a generic tool that checks and graphically displays given executions against a memory model specification. The tool was designed primarily as an aid to hardware designers.

While the above tools are generic, many tools target specific models. Manson and Pugh [20] developed two simulators for the Java memory model that enumerate the possible executions of a program. Java RaceFinder [15], an extension to Java PathFinder [30], is a modern successor. Both of these are explicit-state model checkers. Like CPPMEM (and its predecessor `memevents` [26]), they suffer from the state-explosion problem.

We refer to Batty et al. [4], Torlak et al. [29], and Yang et al. [33] for more related work.

10. Discussion and Conclusion

We applied the model finder Nitpick to our Isabelle/HOL formalization [4] of the C++ draft standard's memory model. Our experiments involved classical litmus tests and (fortunately) did not reveal any flaws in the C++ final draft standard. This is no surprise: The model has already been validated by the CPPMEM simulator on several litmus tests, and the correctness proof of the suggested Intel x86 implementation gave further evidence that the Isabelle model captures the draft standard's intended semantics.

The main challenge for a diagnosis tool such as Nitpick is that users of interactive theorem provers tend to write their specifications so as to make the actual proving easy. In contrast, if the Alloy Analyzer or MemSAT performs poorly on a specification, the tool's developers can put part of the blame on the users, arguing for example that they have "not yet assimilated the relational idiom" [16, p. 7]. We wish we could have applied Nitpick directly on the Isabelle specification of the memory model, but without changes to either Nitpick or the specification our approach would not have scaled to handle even the simplest litmus tests.

We were delighted to see that function specialization, one of the very first optimizations implemented in Nitpick [6, §5.1], proved equal to the task. By propagating arguments to where they are needed, specialization ensures that no more than two arguments ever need to be passed at a call site—a dramatic reduction from the 10 or more arguments taken by many of the memory model's functions. Without this optimization, we would have faced the unappealing prospect of rewriting the specification from scratch.

There will always be cases where more dedicated tools are called for, but it is pleasing when a general-purpose tool outperforms dedicated solutions. Our new Nitpick optimizations will be included in the next Isabelle release and should prove beneficial to other large formalizations.

Acknowledgments

This work would not have been possible without Peter Sewell, who together with the last four authors specified the C++ memory model in Isabelle/HOL. Sascha Böhme, Lukas Bulwahn, Paul Jackson, Tobias Nipkow, Peter Sewell, Mark Summerfield, Geoff Sutcliffe, and the anonymous reviewers suggested several textual improvements. We acknowledge funding from the Deutsche Forschungsgemeinschaft (grant Ni 491/11-2) and the British EPSRC (grants EP/F036345, EP/F067909, EP/H005633, EP/H027351).

References

[1] Programming languages—C++. Technical Report N3290, ISO IEC JTC1/SC22/WG21, 2011. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3290.pdf`.

[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof (2nd Ed.)*, volume 27 of *Applied Logic*. Springer, 2002.

[3] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency: The post-Rapperswil model. Technical Report N3132, ISO IEC JTC1/SC22/WG21, 2010. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf`.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *POPL 2011*, pages 55–66. ACM, 2011.

[5] J. C. Blanchette. Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. *Softw. Qual. J.* To appear.

[6] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

[7] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In R. De Nicola, editor, *ESOP 2007*, volume 4421 of *LNCS*, pages 331–346. Springer, 2007.

[8] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5:56–68, 1940.

[9] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[10] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[12] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[13] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[14] J. Jürjens and T. Weber. Finite models in FOL-based crypto-protocol verification. In P. Degano and L. Viganò, editors, *ARSPA-WITS 2009*, volume 5511 of *LNCS*, pages 155–172. Springer, 2009.

[15] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *ASE 2009*, pages 495–499. IEEE, 2009.

[16] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In H. C. Gall, editor, *ESEC/FSE 2005*. ACM, 2005.

[17] C. Lameter. Effective synchronization on Linux/NUMA systems. Presented at the Gelato Conference 2005.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[19] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *LNCS*, pages 427–447. Springer, 2010.

[20] J. Manson and W. Pugh. The Java memory model simulator. In *Formal Techniques for Java-like Programs (FTfJP) 2002*.

[21] A. McIver and T. Weber. Towards automated proof support for probabilistic distributed systems. In G. Sutcliffe and A. Voronkov, editors, *LPAR 2005*, number 3835 in LNAI, pages 534–548. Springer, 2005.

[22] A. C. Melo and S. C. Chagas. Visual-MCM: Visualising execution histories on multiple memory consistency models. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *ACPC 1999*, volume 1557 of *LNCS*, pages 500–509. Springer, 1999.

[23] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[24] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lightweight tools for heavyweight semantics. In *ITP 2011*. Springer. To appear.

[25] W. Pugh. The Java memory model is fatally flawed. *Concurrency—Practice and Experience*, 12(6):445–455, 2000.

[26] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In Z. Shao and B. C. Pierce, editors, *POPL 2009*, pages 379–391. ACM, 2009.

[27] J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In J. Vitek, editor, *ECOOP 2008*, volume 5142 of *LNCS*, pages 27–51. Springer, 2008.

[28] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

[29] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In B. G. Zorn and A. Aiken, editors, *PLDI 2010*, pages 341–350. ACM, 2010.

[30] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng. J.*, 10(2):203–232, 2003.

[31] T. Weber. A SAT-based Sudoku solver. In G. Sutcliffe and A. Voronkov, editors, *LPAR 2005 (Short Papers)*, pages 11–15, 2005.

[32] T. Weber. *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Dept. of Informatics, T.U. München, 2008.

[33] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS 2004*. IEEE, 2004.